HAL
archives-ouvertes.fr

Generic web service wrapper for efficient embedding of legacy codes in service-based workflows

Tristan Glatard, David Emsellem, Johan Montagnat

▶ **To cite this version:**

Tristan Glatard, David Emsellem, Johan Montagnat. Generic web service wrapper for efficient embedding of legacy codes in service-based workflows. Grid-Enabling Legacy Applications and Supporting End Users Workshop, Jun 2006, Paris, France. pp.1-10, 2006. <hal-00683196>

HAL Id: hal-00683196

https://hal.archives-ouvertes.fr/hal-00683196

Submitted on 28 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Generic web service wrapper for efficient embedding of legacy codes in service-based workflows

Tristan Glatard, David Emsellem, Johan Montagnat
I3S laboratory, CNRS, http://www.i3s.unice.fr/~johan
{glatard,emsellem,johan}@i3s.unice.fr

Abstract

In this paper, we present a generic wrapper that enables the optimization of legacy codes assembled in application workflows on grid infrastructures. We first describe advantages of a service-based approach for job management. We then introduce our wrapper, that works at execution time, thus allowing service grouping strategies to optimize the execution. We demonstrate performance results on a real medical imaging application. We finally propose a new service oriented architecture of the whole system, from application composition to job submission on the grid.

1 Introduction

Grids technologies are very promising for addressing the computing and storage needs arising from many scientific and industrial application areas. In this context, enabling legacy code execution on modern grid infrastructures is challenging. A tremendous amount of work has been put in the development of various sequential data processing algorithms without taking into account particularities of distributed systems nor specific middlewares. Even considering new codes development, instrumenting applications with middleware specific interfaces or designing applications to explicitly take advantage of distributed grid resources is a significant burden for the developers who are often reluctant to allocate sufficient effort on non application specific problems.

Grid middlewares are therefore expected to ease sequential codes migration to grid infrastructures by (i) proposing a non-intrusive interface to existing application code, and (ii) optimizing the execution of the application on grid resources.

In this paper we first discuss task-based job submission and service-based code invocation, the two main paradigms proposed for executing application code on grid infrastructures. We defend the benefit of the service-based approach from a user point of view.

We then propose an application-independent service wrapper to ease the migration of existing application code in the service-based framework at very little cost, thus allowing an easy composition of complex application workflows. Our service wrapper is dynamically treating processing requests at the execution time and can thus be exploited to optimize the application execution on grid infrastructures by grouping jobs, as shown by experimental results. We then propose a complete service-oriented architecture (SOA) of the system, from application composition to grid execution.

2 Grid enabling applications

Two main paradigms are used in different grid middlewares for describing and controlling application processings. The task-based approach is the most widely available and has been exploited for the longest time. It consists of a command-line description and the remote execution of application code. The service-based approach has more recently emerged. It consists in using a standard invocation protocol for calling application code embedded in the service. It is usually completed by a service discovery and an interface description mechanism.

2.1 Task-based job submission

In the task-based job submission approach, each processing is related to an executable code and described through an individual computation *task*. A task description encompasses at least the executable code name and a command-line to be used for code invocation. It may be completed by additional parameters such as input and output files to be transferred prior or next to the execution, and additional task scheduling information such as minimum system requirements. Tasks may be described either directly on the job submission tool command-line, or indirectly through a task description file. Unless considering very simple code invocation use cases, description files are often needed to specify the task in depth. Many file description formats have been proposed and the GGF is currently working on unifying different formats in the Job Submission Descrip-

tion Language (JSDL) working group [1]. The task-based approach is also often referred to as *global computing*.

In the task-based paradigm, code invocation is straightforward, through the legacy code command-line. It does not require any adaptation of the user code and for this reason it has been implemented in most existing batch systems for decades (e.g. PBS, NQS, OAR). Many grid middlewares are also task-based, such as Globus Toolkit, CONDOR, LCG2 and gLite. Indeed, even if those middlewares (Globus Toolkit and gLite in particular) may themselves be made from services orchestrating features such as security, job management and data management, the computing resources of the grid are accessed through task submissions.

2.2 Service-based code execution

The service-based approach was widely adopted for dealing with heterogeneous and distributed systems. In particular for middleware development, the OGSA framework [5] and the subsequent WSRF standard encountered a wide adoption from the international community. In the service-based approach, the code is embedded in a standard service shell. The standard defines an interface and an invocation procedure. The Web Services standard [20], supported by the W3C is the most widely available although many existing implementations do not conform to the whole standard yet. It has been criticized for the low efficiency resulting from using text messages in XML format and alternatives such as GridRPC [15] have been designed to speed-up message exchanges. The service-based approach is also often referred to as *meta computing*. Middlewares such as DIET [3] adopted this approach.

The main advantage of the service based approach is the flexibility that it offers. Clients can discover and invoke any service through standard interfaces without any prior knowledge on the code to be executed. The service-based approach delegates to the server side the actual code execution procedure. However, all application codes need to be instrumented with the service interface to become available. In the case of legacy code application, it is often not the case and an intermediate code invocation layer or some code reworking is needed to exploit this paradigm. Users are often reluctant to invest efforts in writing specific code for services on the application side for different reasons:

- The complexity of standards often makes service conformity a matter of specialists. Some tooling are available for helping in generating service interfaces but they cannot be fully automated and they all require a developer intervention.
- Standards tend to evolve quickly, especially in the grid area, obsoleting earlier efforts in a too short time scale.
- Multiple standards exist and a same application code may need to be executed through different service interfaces.
- In the case of legacy code, recompilation for instrumenting the code may be very difficult or even impossible (in case of non availability of source code, compilers, dependencies, etc).

Therefore, the only way to deal with legacy code in a user-friendly way is to propose a service-compliant code execution interface.

2.3 Discussion

Apart from the invocation procedures and the ease of implementation mentioned above, the task-based and the service-based approaches differ by several fundamental points which impact their usage:

- To submit a task-based job, a user needs to precisely know the command-line format of the executable, taking into account all of its parameters. It is not always the case when the user is not one of the developers. In the service-based approach conversely, the actual code invocation is delegated to the service which is responsible for the correct handling of the invocation parameters. The service is a black box from the user side and to some extent, it can deal with the correct parametrization of the code to be executed.
- The handling of input/output data is very different in both cases. In the task-based approach, input/output data have to be explicitly specified in the task description. Invoking a new execution of a same code on different data requires the rewriting of a new task description. Services better decouple the computation and data handling parts. A service dynamically receives inputs as parameters. This decoupling of processing and data is particularly important when considering the processing of complete data sets rather than single data. Indeed, grid infrastructures are particularly well suited for data-intensive applications that require repeated processings of different data.
- The service-based approach enables discovery mechanisms and dynamic invocation even for *a priori* unknown services. This provides a lot of flexibility both for the user (discovery of available data processing tools and their interface) and the middleware (automatic selection of services, alternatives services discovery, fault tolerance, etc).
- In the service-based framework, the code reusability is also improved by the availability of a standard invocation interface. In particular, services are naturally well adapted to describe applications with a complex workflow, chaining different processings whose outputs are piped to the inputs of each other.
- Services are adding an extra layer between the code invocation and the grid infrastructure on which jobs are

submitted. The caller does not need to know anything about the underlying middleware that will be directly invoked internally by the service. Different services might even communicate with different middlewares and/or different grid infrastructures.

- On the other hand, services deployment introduces an extra effort w.r.t the task-based approach. Indeed, to enable the invocation, services first have to be installed on all the targeted resources, which becomes a challenging problem when their number rises.

The flexibility and dynamic nature of services depicted above is usually very appreciated from the user point of view. Given that application services can be deployed at a very low development cost, there are number of advantages in favor of this approach.

From middleware developers point of view, the efficient execution of application services is more difficult though. As mentioned above, the service is an intermediate layer between the user and the grid middleware. Thus, the user does not know nor see anything of the underlying infrastructure. Tuning of the jobs submission for a specific application is more difficult. Services are completely independent from each other and global optimization strategies are thus hardly usable. Therefore, some precautions need to be taken when considering service based applications to ensure good application performances.

2.4 Workflow of services

Building applications by assembling legacy codes for processing and analyzing data is very common. It allows code reusability without introducing a too high load on the application developers. The logic of such a composed application, referred to as the *application workflow*, is described through a set of computation tasks to perform and data dependencies imposing constraints on the order of processings.

Many workflow representation formats and execution managers have been proposed in the literature with very different properties [21]. When dealing with workflows, the task-based and the service-based paradigms exhibit new fundamental differences.

The emblematic task-based workflow manager is the CONDOR Directed Acyclic Graph Manager (DAGMan) [13]. Based on the static description of such a workflow, many different optimization strategies for the execution have been proposed [2].

Services are naturally very well suited for representing and chaining workflow components. The service based approach has been implemented in different workflow managers such as: the Kepler system [14] which can orchestrate standard Web-Services; the Taverna project [17], from the myGrid e-Science UK project[1] which is able to enact Web-Services and other components such as Soaplab services [18] and Biomoby ones; Triana [19], from the GridLab project[2], which is decentralized and distribute several control units over different computing resources, implementing both a parallel and a peer-to-peer distribution policies; the MOTEUR workflow enactor, developed in our team [7], aims at optimizing the execution of data intensive applications.

Thanks to the decoupling between processings and data, services easily accommodate with input data sets. Data sources are sequentially delivering input data but no additional complexity of the application graph is needed. An example of the flexibility offered by the service-based approach is the ability to define different *data composition strategies* over the input data of a service. When a service owns two inputs or more, a composition strategy defines the composition rule for the data coming from all input ports pairwise. Considering two input sets $\mathrm{A} = \{\mathrm{A}_0, \mathrm{A}_1, \ldots, \mathrm{A}_n\}$ and $\mathrm{B} = \{\mathrm{B}_0, \mathrm{B}_1, \ldots, \mathrm{B}_m\}$ to a service, the most common strategy is a *one-to-one* composition which consists in processing each data of the first set with the matching data of the second set in their order of definition, thus producing $\min(n, m)$ result. This corresponds to the case where a sequence of pairs need to be processed. Another common composition strategy is an *all-to-all* strategy which consists in processing all input data from the first set with all input data from the second set, thus producing $m \times n$ results.

Using iteration strategies to design complex data interaction patterns is a very powerful tool for data-intensive application developers. This is another advantage associated to the service-based approach from the user point of view.

2.5 Legacy code wrapping

To ease the embedding of legacy-codes in the service-based framework, an application-independent job submission service is required. In this section, we briefly review systems that are used to wrap legacy code into services to be embedded in service-based workflows.

The Java Native Interface (JNI) has been widely adopted for the wrapping of legacy codes into services. Wrappers have been developed to automate this process. In [9], an automatic JNI-based wrapper of C code into Java and the corresponding type mapper with Triana [19] is presented: JACAW generates all the necessary java and C files from a C header file and compiles them. A coupled tool, MEDLI, then maps the types of the obtained Java native method to Triana types, thus enabling the use of the legacy code into this workflow manager. Related to the ICENI workflow manager [6], the wrapper presented in [12] is based on code reengineering. It identifies distinct components from a code

[1] myGrid project, `http://mygrid.org.uk`

[2] GridLab project, `http://www.gridlab.org`

analysis, wrap them using JNI and adds a specific CXML interface layer to be plugged into an ICENI workflow.

The WSPeer framework [8], interfaced with Triana, aims at easing the deployment of Web-Services by exposing many of them at a single endpoint. It differs from a container approach by giving to the application the control over service invocation. The Soaplab system [18] is especially dedicated to the wrapping of command-line tools into Web-Services. It has been largely used to integrate bioinformatics executables in workflows with Taverna [17]. It is able to deploy a Web-Service in a container, starting from the description of a command-line tool. This command-line description, referred to as the metadata of the analysis, is written for each application using the ACD text format file and then converted into a corresponding XML format. Among domain specific descriptions, the authors underline that such a command-line description format must include (i) the description of the executable, (ii) the names and types of the input data and parameters and (iii) the names and types of the resulting output data. As described latter, the format we used includes those features and adds new ones to cope with requirements of the execution of legacy code on grids.

The GEMLCA environment [4] addresses the problem of exposing legacy code command-line programs as Grid services. It is interfaced with the P-GRADE portal workflow manager [10]. The command-line tool is described with the LCID (Legacy Code Interface Description) format which contains (i) a description of the executable, (ii) the name and binary file of the legacy code to execute and (iii) the name, nature (input or output), order, mandatory, file or command line, fixed and regular expressions to be used as input validation. A GEMLCA service depends on a set of target resources where the code is going to be executed. Architectures to provide resource brokering and service migration at execution time are presented in [11].

Apart from this latest early work, all of the reviewed existing wrappers are static: the legacy code wrapping is done offline, before the execution. This is hardly compatible with our approach, which aims at optimizing the whole application execution at run time. We thus developed a specific grid submission Web-Service, which can wrap any executable at run time, thus enabling the use of optimization strategies by the workflow manager.

The following section 3 introduces a generic application code wrapper compliant with the Web Services specification. It enables the execution of any legacy executable through a standard service interface. The subsequent section 4 proposes a code execution optimization strategy that can be implemented thanks to this generic wrapper. Finally, section 5 proposes a service oriented architecture of the system, based on a service factory.

3 Generic web service wrapper

We developed a specific grid submission Web Service. This service is generic in the sense that it is unique and it does not depend on the executable code to submit. It exposes a standard interface that can be used by any Web Service compliant client to invoke the execution. It completely hides the grid infrastructure from the end user as it takes care of the interaction with the grid middleware. This interface plays the same role as the ACD and LCID files quoted in the previous section, except that it is interpreted at the execution time.

3.1 Generic web service wrapper

To accommodate to any executable, the generic service is taking two different inputs: a descriptor of the legacy executable command line format, and the input parameters and data of this executable. The production of the legacy code descriptor is the only extra work required from the application developer. It is a simple XML file which describes the legacy executable location, command line parameters, input and output data.

3.2 Legacy code descriptor

The command line description has to be complete enough to allow dynamic composition of the command line from the list of parameters at the service invocation time and to access the executable and input data files. As a consequence, the executable descriptor contains:

1. The name and access method of the executable. In our current implementation, access methods can be a URL, a Grid File Name (GFN) or a local file name. The wrapper is responsible for fetching the data according to different access modes.
2. The access method and command-line option of the input data. As our approach is service-based, the actual name of the input data files is not mandatory in the description. Those values will be defined at the execution time. This feature differs from various job description languages used in the task-based middlewares. The command-line option allows the service to dynamically build the actual command-line at the execution time.
3. The command-line option of the input parameters: parameters are values of the command-line that are not files and therefore which do not have any access method.
4. The access method and command-line option of the output data. This information enables the service to register the output data in a suitable place after the execution. Here again, in a service-based approach, names

of output data files cannot be statically determined because output file names are only generated at execution time.

5. The name and access method of the sandboxed files. Sandboxed files are external files such as dynamic libraries or scripts that may be needed for the execution although they do not appear on the command-line.

3.3 Example

An example of a legacy code description file is presented in figure 1. It corresponds to the description of the service `crestLines` of the workflow depicted in figure 4. It describes the script `CrestLines.pl` which is available from the server `legacy.code.fr` and takes 3 input arguments: 2 files (options `-im1` and `-im2` of the command-line) that are already registered on the grid as GFNs at execution time and 1 parameter (option `-s` of the command-line). It produces 2 files that will be registered on the grid. It also requires 3 sandboxed files that are available from the server.

3.4 Discussion

This generic service highly simplifies application development because it is able to wrap any legacy code with a minimal effort. The application developer only needs to write the executable descriptor for her code to become service aware.

But its main advantage is in enabling the sequential services grouping optimization introduced in section 4. Indeed, as the workflow enactor has access to the executable descriptors, it is able to dynamically create a virtual service, composing the command lines of the codes to be invoked, and submitting a single job corresponding to this sequence of command lines invocation.

It is important to notice that our solution remains compatible with the services standards. The workflow can still be executed by other enactors, as we did not introduce any new invocation method. Those enactors will make standard service calls (e.g. SOAP ones) to our generic wrapping service. However, the optimization strategy described in the next section is only applicable to services including the descriptor presented in section 3.2. We call those services MOTEUR services, referring to our workflow manager presented in section 2.4.

4 Services grouping optimization strategy

The main interest for using grid infrastructures in the processing of data-intensive applications is to exploit the potential application parallelism thanks to the distributed grid resources available. There are three different levels of parallelism that can be exploited when considering any application workflow. Services grouping strategies have to cautiously take care of them, to avoid execution slow down.

```
<description>
  <executable name="CrestLines.pl">
    <access type="URL">
      <path value="http://legacy.code.fr"/>
    </access>
    <value value="CrestLines.pl"/>
    <input name="floating_image" option="-im1">
      <access type="GFN"/>
    </input>
    <input name="reference_image" option="-im2">
      <access type="GFN"/>
    </input>
    <input name="scale" option="-s"/>
    <output name="crest_reference" option="-c1">
      <access type="GFN"/>
    </output>
    <output name="crest_floating" option="-c2">
      <access type="GFN"/>
    </output>
    <sandbox name="convert8bits">
      <access type="URL">
        <path value="http://legacy.code.fr"/>
      </access>
      <value value="Convert8bits.pl"/>
    </sandbox>
    <sandbox name="copy">
      <access type="URL">
        <path value="http://legacy.code.fr"/>
      </access>
      <value value="copy"/>
    </sandbox>
    <sandbox name="cmatch">
      <access type="URL">
        <path value="http://legacy.code.fr"/>
      </access>
      <value value="cmatch"/>
    </sandbox>
  </executable>
</description>
```

Figure 1. Descriptor example

Workflow parallelism. The intrinsic workflow parallelism depends on the application graph topology. For instance if we consider the application example presented in figure 4, services `Baladin` and `Yasmina` can be executed in parallel.

Data parallelism. Data are processed independently from each other. Therefore, different input data can be processed in parallel on different resources. This may lead to considerable performance improvements given the high level of parallelism achievable in data-intensive applications.

Services parallelism. The processing of two different data sets by two different services are totally independent. This pipelining model, very successfully exploited inside CPUs, can be adapted to sequential parts of service-based workflows. Considering the workflow represented on figure 4, services `crestLines` and `crestMatch` may be run in parallel on independent data sets. In practice this kind of parallelism strongly improves the workflow execution on grids.

4.1 Grouping service calls

We propose a services grouping strategy to further optimize the execution time of a workflow. Services grouping consists in merging multiple jobs into a single one. It reduces the grid overhead induced by the submission, scheduling, queuing and data transfers times whereas it may also reduce the parallelism. In particular, sequential processors grouping is interesting because those processors do not benefit from any parallelism. For example, considering the workflow of our application presented on figure 4 we can, for each data set, group the execution of the `crestLines` and the `crestMatch` jobs on the one hand and the `PFMatchICP` and the `PFRegister` ones on the other hand.

Grouping jobs in the task-based approach is straightforward and it has already been proposed for optimization [2]. Conversely, jobs grouping in the service-based approach is usually not possible given that (i) the services composing the workflow are totally independent from each other (each service is providing a different data transfer and job submission procedure) and (ii) the grid infrastructure handling the jobs does not have any information concerning the workflow and the job dependencies. Consider the simple workflow represented on the left side of figure 2. On top, the services for P_1 and P_2 are invoked independently. Data transfers are handled by each service and the connection between the output of P_1 and the input of P_2 is handled at the workflow engine level. On the bottom, P_1 and P_2 are grouped in a virtual single service. This service is capable of invoking the code embedded in both services sequentially, thus resolving the data transfer and independent code invocation issues.

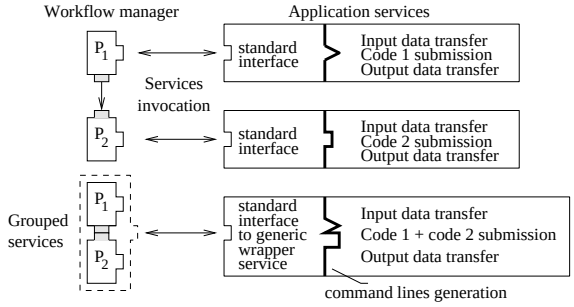

Figure 2. Classical services invocation (top) and services grouping (bottom).

4.2 Grouping strategy

Services grouping can lead to significant speed-ups, especially on production grids that introduce high overheads, as it is demonstrated in the next section. However, it may also slow down the execution by limiting parallelism. We thus have to determine efficient strategies to group services.

In order to determine a grouping strategy that does not introduce any overhead, neither from the user point of view, nor from the infrastructure one, we impose the two following constraints: (i) the grouping strategy must not limit any kind of parallelism (user point of view) and (ii) during their execution, jobs cannot communicate with the workflow manager (infrastructure point of view). The second constraint prevents a job from holding a resource just waiting for one of its ancestor to complete. An implication of this constraint is that if services A and B are grouped together, the results produced by A will only be available once B will have completed.

A workflow may include both MOTEUR Web-Services (*i.e.* services that are able to be grouped) and classical ones, that could not be grouped. Assuming those two constraints, the following rule is sufficient to process all the possible groupings of two services of the workflow:

> Let A be a MOTEUR service of the workflow and $\{B_0,...B_n\}$ its children in the service graph. IF there exists a MOTEUR child B_i which is an ancestor of every B_j ($i \neq j$) and whose each ancestor C is an ancestor of A or A itself, THEN group A and B_i.

Indeed, every violation of this rule also violates one of our constraints as it can easily be shown. The grouping strategy tests this rule for each MOTEUR service A of the workflow. Groups of more than two services may be recursively composed from successive matches of the grouping rule.

For example, the workflow displayed in figure 3, extracted from our medical imaging application, is made of 4 MOTEUR services that can be grouped into a single one through 3 applications of the grouping rule. On this figure, notations nearby the services corresponds to the ones introduced in the grouping rule.

The first application case of the grouping rule is represented on the left of the figure. The tested MOTEUR service A is `crestLines`. A is connected to the workflow inputs and it has two children, B_0 and B_1. B_0 is a father of B_1 and it only has as single ancestor which is A. The rule thus matches: A and B_0 can be grouped. If there were a service C ancestor of B_0 but not of A as represented on the figure, the rules would not match: A and C would have to be executed in parallel before starting B_0. Similarly, if there were a service D the rule would not match as the workflow manager would need to communicate results during the execution of the grouped jobs in order to allow workflow parallelism between B_0 and D.

In the second application case, in the middle of the figure, the tested service A is now `crestMatch`. A has only

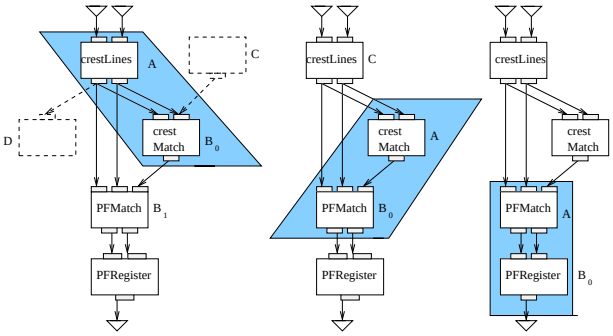

Figure 3. Services grouping examples

a single child: B_0. B_0 has two ancestors, A and C. The rule matches because C is an ancestor of A. A and B_0 can then be grouped. For the last rule application case, on the right of figure 3, A is the `PFMatch` service. It has only one child, B_0, who only has a single ancestor, A. The rule matches and those services can thus be grouped.

When A is the `PFRegister` service, the grouping rule does not match because it does not have any child. Note that in this example, the recursive grouping strategy will lead to a single job submission.

4.3 Experiments on a production grid

To quantify the speed-up introduced by services grouping on a real application workflow, we made experiments on the EGEE production grid infrastructure. The EGEE system is a pool of thousands computing (standard PCs) and storage resources accessible through the LCG2 middleware. The resources are assembled in computing centers, each of them running its internal batch scheduler. Jobs are submitted from a user interface to a central Resource Broker which distributes them to the available resources. The access to EGEE grid resources is controlled for each virtual organizations (VOs). For our VO, about 3000 CPUs accessible through 25 batch queues are available. The large scale and multi-users nature of this infrastructure makes the overhead due to submission, scheduling and queuing time of the order of 5 to 10 minutes. Limiting job submissions by services grouping is therefore highly suitable on this kind of production infrastructure.

4.3.1 Experimental workflows

We made experiments on a medical imaging registration application which is made from 6 legacy algorithms developed by the Asclepios team of INRIA Sophia-Antipolis [16]. The workflow of this application is represented on figure 4. It aims at assessing the accuracy of 4 registration algorithms, namely `crestMatch`, `PFMatchICP/PFRegister`, `Baladin` and `Yasmina`. A number of input image pairs constitutes the input of the workflow (`floating image` and `reference image`). Those pairs are first registered by the `crestMatch` method and this result initializes the 3 remaining algorithms. At the end of the workflow, the `MultiTransfoTest` service is a statistical step that computes the accuracy of each algorithm from all the previously obtained results. `crestLines` is a pre-processing step for `crestMatch` and `PFMatchICP`.

Number of input image pairs	Speed-up on the sub-workflow	Speed-up on the whole application
12	2.91	1.42
66	1.72	1.34
126	2.30	1.23

Table 1. Grouping strategy speed-ups

To show how services grouping is able to speed-up the execution on highly sequential applications, we also extracted a sub-workflow from our application, as shown in figure 4. It is made of 4 services that correspond to the `crestLines`, `crestMatch`, `PFMatchICP` and `PFRegister` ones in the application workflow. Our grouping rule groups those 4 services into a single one, as it has been detailed in the example of figure 3. It is important to notice that even if this sub-workflow is sequential, and thus does not benefit from workflow parallelism, its execution on a grid does make sense because of data and service parallelisms.

To evaluate the impact of our grouping strategy on the performances, we compared the execution times of those workflows with and without using the grouping strategy.

4.3.2 Results

Table 1 presents the speed-ups induced by our grouping strategy for a growing number of input image pairs and for the two experimental workflows described above. This speed-up indicates the acceleration provided by the grouping strategy with respect to a regular grid execution, where each service invocation leads to a job submission. We can notice on those tables that services grouping does effectively provide a significant speed-up on the workflow execution. This speed-up is ranging from 1.23 to 2.91.

The speed-up values are greater on the sub-workflow than on the whole application one. Indeed, on the sub-workflow, 4 services are grouped into a single one, thus providing a 3 jobs submission saving for each input data set. On the whole application workflow, the grouping rule is applied only twice, leading to a 2 jobs saving for each input data set, as depicted on figure 4.

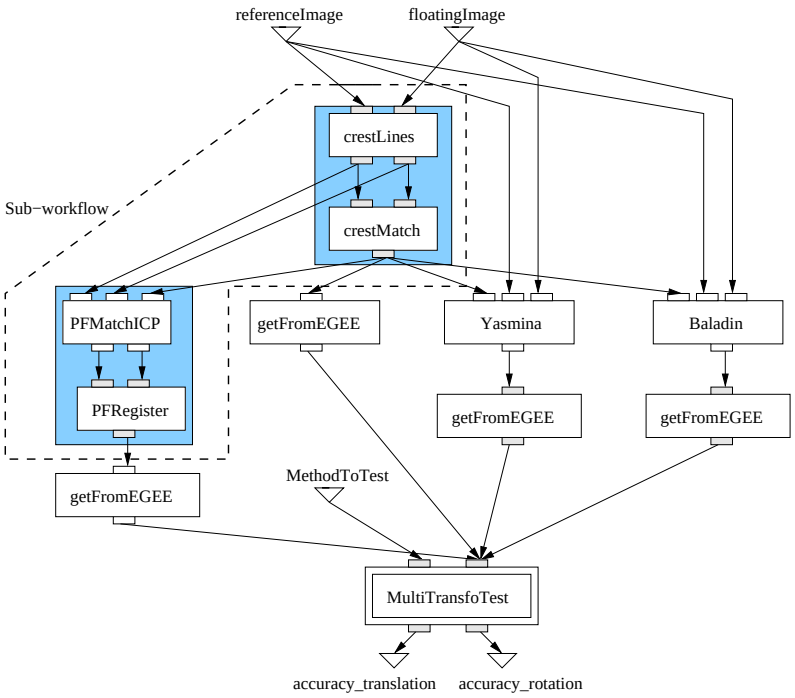

Figure 4. Workflow of the application. Services to be grouped are squared in blue.

5 Dynamic generic service factory

The generic web service wrapper introduced in section 3 drastically simplifies the embedding of legacy code into application services. However, it is mixing two different roles: (i) the legacy command line generation and (ii) the grid submission. Submission is only dependent on the target grid and not on the application service itself. In a Service Oriented Architecture (SOA) it is preferable to split these two roles into two independent services for several reasons. First, the submission code does not need to be replicated in all application services. Second, the submission role can be transparently and dynamically changed (to submit to a different infrastructure) or updated (to adapt to middleware evolutions). In addition, an application wrapper factory service further facilitates the wrapping of legacy code services and their grouping. We thus introduce a complete SOA design based on three main services as illustrated in figure 6.

The (blue) MOTEUR web services represents legacy code wrapping services. They are assembling command lines and invoking the (red) submission service for handling code execution on the grid infrastructure. The code wrapper factory service is responsible for dynamically generating and deploying application services. The aim of this factory is to achieve two antagonist goals:

- To expose legacy codes as autonomous web services respecting the main principles of Service Oriented Architectures (SOA).
- To enable the grouping of two of these web services as a unique one for optimizing the execution.

On one hand, the specific web service implementation details (*i.e.* the execution of legacy code on a grid infrastructure) are hidden to the consumer. On the other hand, when the consumer is a workflow manager which can group jobs, it needs to be aware of the real nature the web service (the encapsulation of a MOTEUR descriptor) so that it could merge them at run time. We choose to use the WSDL XML Format extension mechanism which allows to insert user defined XML elements in the WSDL content itself. On figure 6, we represent the overall architecture and some usage scenario. First, the legacy code provider submits (**1.a**) a MOTEUR XML descriptor P1 to the MOTEUR factory. The factory, then dynamically deploy (**1.b**) a web service which wraps the submission of the legacy code to the grid via the generic service wrapper. Another provider do the same with the descriptor of P2 (**2.a**). The resulting web services expose their WSDL contracts to the external world with a specific extension associated with the WSDL operation. For instance, the WSDL contract resulting of the deployment of the `crestLines` legacy code described on figure 1 is printed on figure 5.

This WSDL document defines two types (`CrestLines-request` and `CrestLines-`

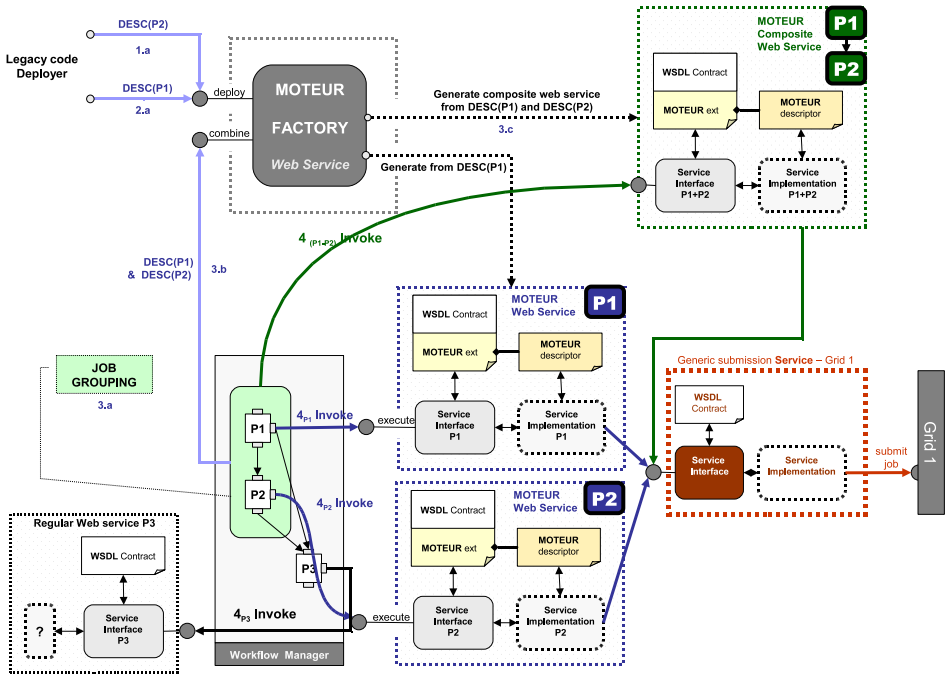

Figure 6. Services factory.

`response`) corresponding to the descriptor inputs and outputs and a single `Execute` operation. Notice that in the binding section, the WSDL document contains an extra `MOTEUR-descriptor` tag pointing to the URL of the legacy code descriptor file (`location`) and a binding to the Execute operation (`soap:operation`).

Suppose now that the workflow manager identifies a services grouping optimization (*e.g.* P1 and P2) (**3.a** on figure 6). Because of its ability to discover the extended nature of these two services, the engine can retrieve the two corresponding MOTEUR descriptors. It can ask the factory to *combine* them (**3.b**) resulting in a single composite web service (**3.c**) which exposes an operation taking its inputs from P1 (and P2 inputs coming from other external services) and returning the outputs defined by P2 (and P1 outputs going to other external services). This composite web service is of the same type than any regular legacy code wrapping service. It is accessible through the same interface and it also delegates the grid submission to the generic submission web service by sending the composite MOTEUR descriptor and the input link of P1 and P2 in the workflow.

6 Conclusion

In this paper, we first described an application-independent legacy code wrapper that works at run time, by interpreting a command-line description file. This wrapper made possible the deployment of a real medical imaging application in a user-friendly way.

We then introduced a workflow optimization strategy based on this wrapper, which consists in grouping services that do not benefit from any parallelism. We showed results on workflows related to our application, deployed on the EGEE infrastructure. Our grouping strategy is able to provide speed-ups close to 3 on one of our examples.

We finally introduced a fully SOA compliant architecture of the whole system, from application composition to job submission, that fully automatizes the legacy code wrapping and the grouping strategy procedures. Any legacy code-based application can thus be instantiated by only providing a MOTEUR descriptor to the service factory.

Acknowledgments

This work is partially funded by the French research program “ACI-Masse de données” (http://acimd.labri.fr/), AGIR project (http://www.aci-agir.org/). We are grateful to the EGEE European project for providing the grid infrastructure and user assistance.

References

[1] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) Specification, Version 1.0. Technical report, GGF, nov 2005.

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions ...>
 <types>
  <schema>
   <element name="CrestLines-request">
     <complexType>
      <sequence>
       <element name="floating_image"
                type="string"... />
       <element name="reference_image"
                type="string"... />
       <element name="scale" type="string"... />
      </sequence>
     </complexType>
    </element>
    <element name="CrestLines-response">
      <complexType>
        <sequence>
          <element name="crest_reference"
                   type="string"... />
          <element name="crest_floating"
                   type="string"... />
        </sequence>
      </complexType>
    </element>
  </schema>
 </types>
 <message name="ExecuteSoapIn">
  <part name="parameters"
        element="CrestLines.pl-request" />
 </message>
 <message name="ExecuteSoapOut">
  <part name="parameters"
        element="CrestLines.pl-response" />
 </message>
 <portType name="CrestLines.plSoap">
  <operation name="Execute">
   <input message="ExecuteSoapIn" />
   <output message="ExecuteSoapOut" />
  </operation>
 </portType>
 <binding ...>
  <soap:binding transport="http://..." />
   <operation name="Execute">
    <soap:operation soapAction="http://.../Execute"
          style="document" />
     <MOTEUR-descriptor xmlns="urn:...">
      <location>http://...</location>
     </MOTEUR-descriptor>
     ....
   </operation>
 </binding>
</definitions>
```

Figure 5. WSDL generated by the factory

[2] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task Scheduling Strategies for Workflow-based Applications in Grids. In *CCGrid*, Cardiff, UK, 2005.

[3] E. Caron and F. Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 2005.

[4] T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, and P. Kacsuk. GEMLCA: Running Legacy Code Applications as Grid Services. *Journal of Grid Computing (JGC)*, 3(1-2), 2005.

[5] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.

[6] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI : Optimisation of component applications within a Grid environment. *Journal of Parallel Computing*, 28(12):1753–1772, 2002.

[7] T. Glatard, J. Montagnat, and X. Pennec. An optimized workflow enactor for data-intensive grid applications. Technical Report I3S/RR-2005-32-, I3S, Sophia-Antipolis, oct 2005.

[8] A. Harrison and I. Taylor. Dynamic Web Service Deployment Using WSPeer. In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 11–16, feb 2005.

[9] Y. Huang, I. Taylor, D. M. Walker, and R. Davies. Wrapping Legacy Codes for Grid-Based Applications. In *17th International Parallel and Distributed Processing Symposium (IPDPS)*, page 139. IEEE Computer Society, 2003.

[10] P. Kacsuk, G. Dzsa, J. Kovcs, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombs. P-GRADE: A Grid Programing Environment. *Journal of Grid Computing (JGC)*, 1(2):171–197, 2003.

[11] G. Kecskemeti, Y. Zetuny, T. Kiss, G. Sipos, P. Kacsuk, G. Terstyanszky, and S. Winter. Automatic deployment of Interoperable Legacy Code Services. In *UK e-Science All Hands Meeting*, Nottingham, UK, sep 2005.

[12] J. Li, Z. Zhang, and H. Yang. A Grid Oriented Approach to Reusing Legacy Code in ICENI Framework. In *IEEE International Conference on Information Reuse and Integration (IRI'05)*, Las Vegas, USA, aug 2005.

[13] M. Livny. Direct Acyclic Graph Manager (DAGMan). http://www.cs.wisc.edu/condor/dagman/.

[14] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*, 2005.

[15] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. Technical report, Global Grid Forum (GGF), jul 2005.

[16] S. Nicolau, X. Pennec, L. Soler, and N. Ayache. Evaluation of a New 3D/2D Registration Criterion for Liver Radio-Frequencies Guided by Augmented Reality. In *International Symposium on Surgery Simulation and Soft Tissue Modeling (IS4TM'03)*, volume 2673 of *LNCS*, pages 270–283, Juanles-Pins, 2003. INRIA Sophia Antipolis, Springer-Verlag.

[17] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics journal*, 17(20):3045–3054, 2004.

[18] M. Senger, P. Rice, and T. Oinn. Soaplab - a unified Sesame door to analysis tool. In *UK e-Science All Hands Meeting*, pages 509–513, Nottingham, sep 2003.

[19] I. Taylor, I. Wand, M. Shields, and S. Majithia. Distributed computing with Triana on the Grid. *Concurrency and Computation: Practice & Experience*, 17(1–18), 2005.

[20] W. World Wide Web Consortium. Web Services Description Language (WSDL) 1.1, mar 2001. http://www.w3.org/TR/wsdl.

[21] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *ACM SIGMOD Record*, 34(3):44–49, Sept. 2005.